AFRL-RI-RS-TR-2014-011

AN XDATA ARCHITECTURE FOR FEDERATED GRAPH MODELS AND MULTI-TIER ASYMMETRIC COMPUTING

SYSTAP, LLC

JANUARY 2014

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2014-011 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
EDWARD L. DEPALMA
Work Unit Manager

/ S /
MICHAEL J. WESSING
Deputy Chief, Information Intelligence
Systems and Analysis Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JAN 2014	FINAL TECHNICAL REPORT	OCT 2012 – NOV 2013

4. TITLE AND SUBTITLE

AN XDATA ARCHITECTURE FOR FEDERATED GRAPH MODELS AND MULTI-TIER ASYMMETRIC COMPUTING

5a. CONTRACT NUMBER
FA8750-13-C-0002

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62702E

6. AUTHOR(S)

MICHAEL PERSONICK
BRYAN THOMPSON

5d. PROJECT NUMBER
XDATA

5e. TASK NUMBER
A0

5f. WORK UNIT NUMBER
14

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SYSTAP, LLC
4501 TOWER RD
GREENSBORO, NC 27410

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIEA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2014-011

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# **88ABW-2014-0162**
Date Cleared: **21 January 2014**

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Scalable, data-parallel graph analytics on GPUs is a fundamentally hard problem that goes beyond the current state of the art. Scalable graph analytics are critical for a large range of application domains with a vital impact on both national security and the national economy. CPU graph algorithms are known to scale poorly due to non-locality and limited memory bandwidth. Our research shows that GPUs provide a high-performance, data-parallel, commodity hardware platform for graph analytics.

15. SUBJECT TERMS

Graphs, Graph Analytics, Big Data, GPU, Many-Core, High-Performance Computing

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**EDWARD L. DePALMA**
U	U	U	UU	40	19b. TELEPHONE NUMBER (*Include area code*) **(315) 330-3069**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 SUMMARY

Scalable, data-parallel graph analytics on many-core hardware is a fundamentally hard problem that goes beyond the current state of the art. Scalable graph analytics are critical for a large range of application domains with a vital impact on both national security and the national economy, including: counter-terrorism; fraud detection; drug discovery; cyber-security; social media; logistics and supply chains; e-commerce, etc. CPU graph algorithms are known to scale poorly due to non-locality and limited memory bandwidth. Our research shows that GPUs provide a high-performance, data-parallel, commodity hardware platform for graph analytics.

Our goal is to develop a scalable, open-source solution for high-performance graph analytics on GPUs. Our approach combines a high-level abstraction that allows analysts to easily write graph analytics that leverage GPUs; a high-performance, data-parallel runtime for the GPU; and a scalable architecture for GPUs and GPU clusters. Our team delivered an initial Thrust-based version of the "GAS Engine" Proof Of Concept (POC) in June. Since then we have re-implemented the graph engine from the ground up using a data-parallel runtime strategy based on leading-edge research [Merrill2012] and released the code under an open-source project, "MPGraph".

We demonstrate GPU scaling on several data sets of interest to DARPA, including Wikipedia, a scale-free random graph (kron), Akamai trace route data, Bitcoin transaction data, and a Twitter follower network. We present results for Breadth First Search (BFS), a fundamental primitive for graph traversal, and Single Source Shortest Path (SSSP). We measure GPU speedups of between 3x (SSSP on a random graph) and nearly 300x (Akamai and Bitcoin) over the CPU performance of a well-known and widely deployed CPU-based graph mining platform that uses a similar high-level abstraction (GraphLab). We also measure significant speedups over our initial POC as shown in **Figure 1**.

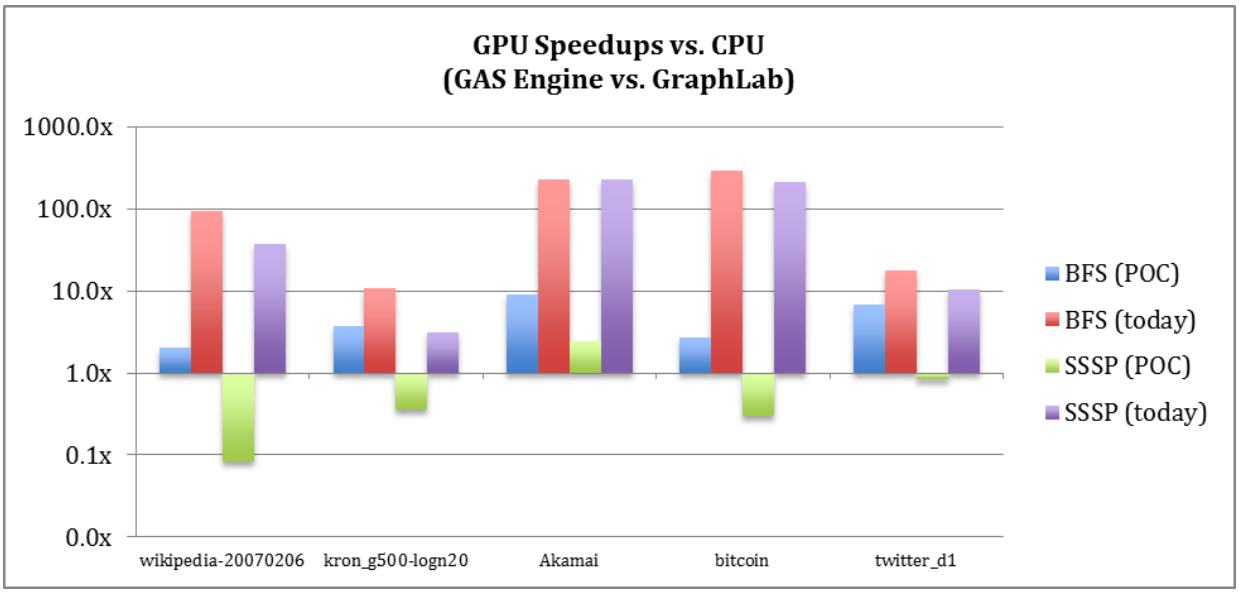


Figure 1: GPU Speedups vs. CPU

Our research in the first year has proven the feasibility of a high-performance graph programming engine on the GPU for two key graph algorithms. However, a single GPU has only 6G of fast device RAM. Therefore, larger graphs must be partitioned to scale off the device. Further, developing code for the GPU is notoriously difficult. Continuing into the second year of research we will:

(a) extend these results to additional graph algorithms;
(b) encapsulate these results within an easy-to-use and extensible open source platform; and
(c) demonstrate that these results can be scaled to multiple GPUs.

2.0 INTRODUCTION

[Merrill2012] demonstrated that GPU can deliver 3 billion Traversed Edges Per Second (3 Giga-TEPS or GTEPS) across a wide range of graphs on Breadth First Search (BFS), a fundamental building block for graph algorithms. Merrill found that the GPU enjoyed a speedup of 12x over the *idealized* multi-core scaling of a 3.4GHz Intel Core i7 2600K CPU (the equivalent of 3 such 4-core CPUs) across the majority of the graphs. Thus, assuming perfect scaling, the throughput of a single GPU is comparable to that of between 12 CPU cores. In fact, (a) CPU graph algorithms are known to have sub-linear scaling; and (b) the GPU performance was significantly higher on some data sets. Since a workstation can host up to 4 GPUs, there is a tantalizing possibility of achieving, in a single workstation, the throughput of a cluster with between 48+ cores (e.g., 6+ servers, each having 8 cores per machine). This suggests that sophisticated, high-performance analytic capabilities could fit under a desk, be delivered on a ship, or forward deployed.

Merrill estimated multi-core CPU scaling in two ways. First, he directly compared with the best published results for multi-core CPU algorithms. Second, he implemented a single-core version of the algorithms, verified performance against published single-core results, and then used idealized linear scaling to estimate multi-core performance. The second approach deliberately hedges performance in favor of the CPU. If fact, as we show below, at least one widely deployed CPU graph mining solution does not scale well as a function of the number of CPU cores. Thus, a GPU enjoys a very significant advantage for graph algorithms over a CPU, at least for a single machine. The main reason for the high performance of the GPU on graph algorithms is the high bandwidth of the device memory. Graph algorithms are memory bound. CPU architectures have slower memory, hit the memory bandwidth bottleneck sooner, and cannot scale beyond that bottleneck by adding more CPU cores.

A single GPU can hold a graph with a billion edges in its high speed DRAM. Scaling to larger graphs requires partitioning the problem. To the best of our knowledge, no published work has demonstrated good scaling to multiple GPUs and GPU clusters on BFS, the fundamental building block of graph algorithms, let alone across a wide range of graphs, algorithms, and data scales – we analyze some reasons for this in the section on Related Work.

Our effort will deliver a highly scalable solution for graph processing on GPUs and GPU clusters that will advance the state of the art.

Our solution will provide:

(1) A *high-level abstraction* for expressing data-parallel graph algorithms: This abstraction will make it possible for ordinary programmers to leverage data-parallel evaluation of graph algorithms on GPUs;

(2) A *data-parallel runtime*: An optimized, data-parallel runtime will deliver the potential of GPUs for graph analytics; and

(3) *Scalable graph analytics that go beyond the state of the art*. Large graphs will be automatically decomposed into patches. Operations on large graphs will be decomposed into tasks that operate over those patches. The patches and tasks will be distributed across the resources of a multi-GPU workstation or GPU-enabled compute cluster in a way that minimizes the communications volume. A local scheduler on each node will run tasks as their prerequisites are satisfied, optimize the bandwidth utilization of the PCIe bus by intelligent data and task placement, and overlap data movement with computation to hide latency.

2.1 Many-Core Architectures

CPU clock rates have been flat for nearly a decade. In order to increase throughput, applications must rely on parallel processing architectures (either large shared memory machines or horizontal scaling on clusters), main memory, and many-core architectures (GPUs, Xeon Phi). The current and next generation of CPU architectures, e.g., Haswell and Broadwell, both integrate GPU processing units into the CPU. This trend will continue since clock rates for CPUs can no longer be increased due to fundamental manufacturing and energy dissipation limits. However, it is a non-trivial problem to scale applications onto these hybrid architectures. In particular, GPU algorithms are hard. They require significant expertise to develop, intimate knowledge of the CPU and GPU memory systems, and detailed knowledge of the Compute Unified Device Architecture (CUDA).

Researchers have known for a decade that *memory bandwidth*, not processor speed, was the primary performance limitation for data intensive applications [BONCZ1999]. While the clock rates for GPUs are much slower than those for CPUs, GPUs have nearly *ten times* the *compute throughput* when compared to modern CPUs (e.g., 1331 single-precision GFLOPS versus 100 single-precision GFLOPS). Further, GPUs also have nearly *ten times* the *memory bandwidth* of modern CPUs (192 GB/s versus 21 GB/s). (Both the FLOPS and the memory bandwidth numbers are for the GTX-580 GPU and the i7-2600 CPU). GPUs are potentially much faster than CPUs for applications that are limited by either compute (FLOPS) or memory bandwidth. The GPU maintains its order of magnitude bandwidth advantage over the CPU for sequential access, random access and Compare And Swap (CAS) operations. Thus,

while coalesced memory access patterns are much faster than random access patterns, the GPU still out performs the CPU by the same margin when non-coalesced access patterns dominate a computation.

Modern GPUs have up to 6GB of high bandwidth memory on the device. Applications that exceed this memory limit need to scale "out of core" – techniques for doing this are discussed below. The GPU can access main memory at the bandwidth of the PCIe bus. A 16-lane PCIe 2.0 bus has peak 8 GB/s one way. PCIe 3.0 doubles this to 16 GB/s. However, out of core scaling can still provide significant performance gains if the application can overlap data movement with computation and benefit from increased memory bandwidth on the GPU. For example, a performance improvement between 3x and 8x has been demonstrated for hash joins on GPUs when scaling out of core [KALDEWAY2012] (the performance gain increases with the size of the join since data transfer costs are amortized). If those transfers can be made to overlap with computation, then the latency of the transfers can be hidden as well.

Today, the world's fastest supercomputers rely on many-core architectures. For example, ORNL Titan (http://www.olcf.ornl.gov/titan/), is a collection of 18,688 compute nodes. While each node has 16 CPU cores, Titan gets most of its 20+ petaflop performance from an NVIDIA K20 GPU on each compute node. Titan, which took first place in late 2012, was surpassed in 2013 by Tianhe-2 ("Milky Way-2"). Tianhe-2 uses 16,000 nodes, each with two Intel Xeon IvyBridge processors and three Intel Xeon Phi processors (the Xeon Phi is a many-core architecture that puts many Pentium class CPUs onto a daughter card).

2.2 Scalable Design

There are three major aspects to a scalable data-parallel application. The application infrastructure must: (a) provide a domain-specific, high-level abstraction for writing applications; (b) provide an efficient, low-level, data-parallel runtime for the domain specific operations; and (c) decompose the problem into tasks, organize those tasks into directed task graphs that expose the maximum amount of parallel work, and overlap computation with data movement. Application code is written using the domain-specific abstraction as a series of tasks. Those tasks are compiled into a form which is executed by a distributed runtime system. On each node, tasks execute using the low-level data-parallel runtime to perform their work efficiently. This approach increases user productivity, inherently future-proofs the architecture, and scales gracefully.

This approach to scalable data-parallel applications is based on lessons learned from large-scale parallel scalability studies with the MIT open-source Uintah Software (http://www.uintah.utah.edu). Uintah has used variants of this approach since 1998. Since 2005, using this approach, Uintah has been extended to run on, and scale to, the very largest machines – this work was performed in a team led by Dr. Berzins, an academic member of the SYSTAP team. Today, the Uintah simulations scale to the world's largest supercomputers, including the ORNL Titan supercomputer. Uintah insulates the application from the rapid evolution of hardware and architectures through

a variety of high-level abstractions, including: (a) *a domain specific abstraction* for writing analytics [Nebo2012]; (b) a *low-level, data-parallel runtime* supporting those analytics; and (c) a *data warehouse* on each multi-core or GPU node which abstracts away hardware specific operations required to support the movement of data and scheduling of tasks in order to hide latency and maximize throughput [Concurrency2012].

Since 2011, Uintah applications are able to run in hybrid computing environments with a mixture of different technologies due, in part, to the future-proofing aspects of the domain-specific abstraction [WOLFHPC, XSEDE2013, Nebo2012]. In order to achieve good per-core or per-GPU efficiency, Uintah uses an approach called Wasatch to decompose each task in the Uintah task-graph still further into a sub-graph that is dynamically executed on a core or a GPU [EUROPAR]. This approach has been shown to produce scalability to 250K cores [CCGRID13] and has recently been found to produce automatically generated code that is an order of magnitude faster than the original hand-written code in the Uintah flow component.

While graph models of computations arise naturally in many situations, there are many significant differences in the structure of the graphs related to the application. In Uintah, graphs are somewhat coarse-grained in nature as they arise from mesh-patch-based computation models of complex systems of partial differential equations. However, the graphs considered here have less regular structure, less locality in their communications, and more complex data types. Thus, the architectural lessons and design principles learned from Uintah can be generalized to graphs, but the task decomposition method and high-level abstraction cannot be directly shared between graphs and these other domains. There are tools available to partition graphs, such as Par Metis [KARYPIS1999] and Zoltan [DEVINE2006, DEVINE2002], and such methods may be used to coarsen and decompose graphs in a way that either approximates or directly minimizes the communication volume. A variety of approaches have been suggested - see Related Work. In addition, experiences with adaptive meshes in Uintah have provided insight into how to address situations in which the workload and the underlying graph structure change in a dynamic way [IPDPS10].

Present work in Uintah is focused on ensuring that multiple GPUs may be used in an efficient manner when there is significant communication between these GPUs. This work has led to the development of new Uintah GPU task APIs that make it easier for component developers to write GPU task kernels. The new Uintah GPU interfaces include GPU *data warehouses* that are similar to the CPU data warehouses previously deployed. The data warehouse works as a combined memory manger, data transport agent, and task scheduler. To do this, the data warehouse maintains a dictionary of required variables and other dependencies for a task to execute and a simple queue structure that is used to monitor when all the information for a task is available [SC2013]. The data warehouse uses this information to manage the movement of data both onto, and off of, the compute device, and to store the results of each task in a way that makes these results available to successive tasks, whether local or remote. The data movements overlap the computations on the compute device in order to keep the

compute device busy as much as possible. Because the data warehouse knows the actual resource requirements for the current and future tasks, it can optimize data movement with respect to the computation. Since the device memory is now managed by the data warehouse, variables can now exist beyond a task kernel execution, thus reducing PCIe memory bandwidth usage. In addition, when there are multiple GPUs in a single node, the GPU data warehouse knows the variables location across those devices so GPUDirect could be used to avoid memory copy through host [WOLFHPC] or to place a task on the device that requires the least data movement. Once the requirements for a given task are satisfied, the Uintah framework will do the CudaMalloc/H2D/D2H memory copy asynchronously and automatically. In this way, the data movement can overlap with GPU kernel execution, CPU task execution, and MPI communications to hide latency. As each Uintah GPU task use one CUDA stream, a kernel will be launched when its required variables are ready in the device memory and multiple kernels and two-way memory copies can be executed concurrently. One the GPU task is launched, the component developer can call a "get" function to find the pre-loaded variable location in the device memory using the variable name, patch ID and an index key.

2.3 High Level Abstraction

Several abstractions have been developed for graph pattern matching and graph algorithms. These abstractions fall into three categories:

- Declarative Query Languages, such as SPARQL [SPARQL2008], GraphQL [HE2008], and Cypher (http://www.neo4j.org). See [WOOD2012] for a recent survey;
- Domain Specific Languages (DSLs), such as GreenMarl [Hong2012]; and
- "Vertex-centric" abstractions, such as Pregel [Pregel2010] , Signal/Collect [Stutz2010], and GraphLab [GraphLab2010, GraphLab2012, PowerGraph2012, GraphChi2012, GraphLab2013].

Graph query languages lend themselves to graph pattern matching and path expressions, but popular graph query languages cannot be used to express iterative graph traversal algorithms such as BFS, SSSP, Connected Components, etc. The Signal/Collect team has demonstrated (private communication) that good performance on graph query languages can be obtained by translating the query into a data flow over the graph. Domain specific languages can be used to write graph algorithms, but DSLs for production systems must be carefully designed in order to allow programmers to go around the abstraction when the DSL implementation encounters a translation problem that is poorly handled by its rule set. The most developed DSL for graphs is GreenMarl, which is being migrated into a high-end Oracle product. However, there has been, as yet, little interest in replicating this approach in the open source community. The "vertex-centric" abstraction has captured the interest of both academic and open source communities with a variety of open source platforms, including GraphLab, Apache Giraph, GoldenOrb, etc. A large set of graph algorithms exists for this approach, and,

even with the lack of a standard, algorithms can often be ported to new platforms with very little development effort. Further, the vertex-centric abstraction makes it relatively easy to develop new algorithms. The SYSTAP team has direct experience with the SPARQL graph query language and includes a high performance SPARQL query engine in their graph database platform.

The GAS abstraction supports 2D partitioning (vertex cuts), which is known to minimize the communication volume [Checconi2012, Vastenhouw2005]. The GAS abstraction also imposes some constraints on the phases in the computation when the vertex and edge state may be accessed in read-only and read/write modes that are known to facilitate scaling. Further, using the GAS abstraction significantly reduces the time required to write new graph analytics for the GPU. For example, it took only hours to implement and test a Connected Components algorithm using the GAS abstraction. In contrast, a low-level implementation of a Connected Components algorithm took several months to develop. We have also shown that, for at least some graph algorithms, e.g., BFS (which requires the traversal of only the out-edges) and Single Source Shortest Path (SSSP) (which requires the traversal of both in-edges and out-edges), the GAS abstraction can be mapped onto the underlying GPU hardware without sacrificing any efficiency. Thus, we believe that this high-level abstraction will significantly reduce the time to develop new graph analytics, obviate the requirement for an expert CUDA engineer to make write those graph analytics, and open up the capabilities of the GPU hardware to a broad base of developers and analysts.

2.4 Data-parallel Runtime

In Uintah, there is an inherent spatial topology that is not present in more general graphs problems. The lack of a spatial interpretation gives rise to non-local access patterns. Graphs also require data-dependent parallelism since the number of edges into and out of each vertex is a vertex specific property that can vary widely both within a single graph and across different graphs. Therefore, operations on graphs require an adaptive, data-dependent parallelism that is significantly more irregular in structure than that seen with Uintah. This irregularity shows up at several levels.

- Thread assignment: [Merrill2012] demonstrated that an adaptive thread assignment approach significantly outperforms a fixed policy. In order to achieve a high throughput on a GPU, the strategies for assigning Threads, Warps, and CTAs to vertices and edges must take into consideration the fan-in and fan-out of each vertex. This data-dependent characteristic varies on a vertex by vertex basis within the graph and different graphs may have characteristically different degree distributions. A similar data-dependent approach is adopted in Uintah to address the execution of a very rich mix of tasks [Concurrency2012].
- Size of the frontier: Some graph algorithms explore a dynamic frontier while others must visit all vertices and edges in the graph in each iteration. Different techniques are more efficient depending on which visitation pattern is used. For example, the Parallel Sliding Window [GRAPHCHI2012] approach provides an IO efficient solution when all vertices must be visited in each iteration. Techniques that produce a

compact frontier can perform significantly better when the size of the frontier is small, but may be dominated by other techniques as the size of the frontier grows. Under some cases, it can become substantially more efficient to reverse the direction of the traversal [BEAMER2012].

- Data layout and compression: GPU graph primitives are constrained by the available memory bandwidth. The overall throughput can be increased by optimizing the memory layout. However, the best memory layout is a data dependent property of the graph related to the static and dynamic aspects of the computation state for the edges and vertices in the graph. The memory layout can be optimized in advance for specialized problems, such as BFS, but a more general approach is needed for a general purpose graph processing framework. Compression can improve the overall throughput by doing more work per byte. Compression can also increase the size of the graph, or graph partition, that fits into the device memory of the GPU. This directly effects the efficiency and scaling of graph computations.
- Approximation: Many algorithms, including betweenness centrality, triangle counting, page rank, etc. can be approximated with an enormous savings in time and memory when compared to their exact computation. Approximate solutions are often examples of working smarter, not harder. We are exploring ways in which techniques for approximation, such as sampling edges [Brandes2001] and computations on reduced rank approximations of the graph [Zhao2013], can be introduced into the high-level abstraction and the data-parallel runtime. When solutions are approximate, either the estimation error associated with the approximation needs to be reported back to the analyst, or the estimation error needs to be used to guide an adaptive resolution in which the computation does a minimum amount of work to ensure a target estimation error in the result (similar to adaptive resolution in Uintah).
- Breaking encapsulation: Some graph algorithms break the encapsulation of the GAS abstraction. Examples include algorithms that require explicit control of synchronization barriers, algorithms that need to maintain an auxiliary data structure (this is often finessed by adding “fake” vertices not present in the given graph in order to capture additional state that must be visible beyond a given vertex or partition), or algorithms that must carry out operations on the graph that are not currently supported by the abstraction (such as sampling or graph mutation). Such algorithms currently must be written to a lower level of the runtime. More research is needed to ensure that the abstraction does not get in the way of writing scalable applications and that lower level abstractions remain available in a distributed parallel architecture.
- Synchronization barriers: Global synchronization can cause tremendous overhead in a large distributed computation since all nodes block during coordination. A variety of techniques can ameliorate those costs, including asynchronous execution [Pearce2010], speculative execution, interleaved computations, and using tree structured communication paths to reduce synchronization latency. All of this should be hidden from the user.

Existing research has shown that GPUs can provide excellent throughput and price/performance for graph problems. However, efficient, data-parallel execution of graph algorithms on multiple GPUs and GPU clusters remains a hard problem requiring innovative and adaptive techniques. Further, high-level abstractions are necessary to expose the performance of the GPU to everyday programmers. By choosing the right abstractions, we can future-proof the application against continuing evolution in hardware architectures. While good scaling has been shown on CPU-based architectures, there has not yet been a serious effort to develop a scalable graph processing architecture for GPUs. We propose to address this fundamental and important research problem.

2.5 Related Work

Graph traversal and graph partitioning have been studied extensively in the literature. On a distributed parallel computer architecture, scalable and efficient graph analytics requires a suitable decomposition of the data and the associated work. The data decomposition is typically done by graph partitioning. Hence, the goal of the graph partitioning is to distribute data and work evenly among processors in a way that reduces communication cost. There are many graph partitioning strategies proposed in the literature. The simplest strategies is the 1D partitioning, which partitions the graph vertices into disjoint sets and assigns each set of vertices to a node. 1D partitioning is implemented in widely used packages such as Metis [KARYPIS1995] and Zoltan [DEVINE2006, DEVINE2002]. These implement 1D partitioning using an efficient multilevel bipartitioning algorithm, which is parallelized by Par Metis [KARYPIS1999]. In 2D partitioning the graph edges are distributed among the compute nodes by arranging the edges into blocks using vertex identifier ranges. These blocks are organized into an n x n grid and mapped onto p virtual processors, where p is a power of two. Each row in the grid contains all in-edges for a range of vertices. The corresponding column contains the out-edges for the same vertices. In [Checconi2012], the authors propose a highly scalable 2D graph partitioning algorithm. They implement BFS with this algorithm on IBM Blue Gene/P and Blue Gene/Q machines using optimizations to reduce communications by 97.3% (through a “wave” propagated along the rows of the 2D partitioning to eliminate duplicate updates) and also optimize for the underlying network topology. This approach was ranked 1st and 3rd in the June 2013 Graph500 (http://www.graph500.org). Vastenhouw and Bisseling [Vastenhouw2005] introduce a distributed method for parallel sparse-matrix multiplication based on 2D graph partitioning. Many of the 1D and 2D Graph partitioning algorithms perform partitioning on the original graphs and try to minimize the edge-cuts. However, it has been shown that for many problems, this edge-cut metric is not an accurate representation of communication cost, and hypergraphs more accurately model the communication cost [Catalyurek1999]. Several open source libraries exists for hypergraph partitioning, such as PaToH [Catalyurek1999-patoh] and hMETIS [Karypis1997]. However, these libraries run in serial. For large-scale parallel applications, partitioning must be performed in parallel. Devine et al. introduce a parallel hypergraph partitioning strategy in [Devine2006] and develop a parallel software package at Sandia National Labs. [PowerGraph2012] and [GraphChi2012] have been shown to be equivalent to 2D

partitioning. Recently, other graph partitioning strategies are proposed for efficient data and work distribution, such as streaming graph partitioning [Stanton2012] and dynamic graph partitioning [Yang2012].

Several methods and software packages are introduced in the literature to use these graph partitioning strategies to develop scalable, high performance graph analytics on parallel architectures. In [Chen2012], the authors try to address the problem of how graph partitioning can be effectively integrated into large graph processing in the cloud environment by developing a novel graph partitioning framework to improve the network performance of graph partitioning itself, partitioned graph storage, and vertex-oriented graph processing. Also, in [Berry2007], Berry el al. introduce the Multi-Threaded Graph Library (MTGL), generic graph query software for processing semantic graphs on multithreaded computers. [Bader2008] introduces a parallel graph library (SNAP). [Agarwal2010] presents results for BFS on the Intel Nehalem EP and EX processors for up to 64 threads in a single system and presents performance comparisons to the Cray XMT [Ediger2013], Cray MTA-2 [Bader2006], and Blue Gene/L. [Pearce2010] presents results for multi-core scaling using asynchronous methods (as opposed to bulk synchronous) for main memory, semi-external memory, and external memory. [Buluc2011] and [Lugo2012] present an approach to graph processing based on sparse matrix-vector operations.

Graph algorithms are memory bound, thus memory layouts (by increasing locality of reference) and data compression (by doing more work per byte) can both improve throughput. [Checconi2012] discusses the use of compression to improve locality of reference for BFS on large clusters, techniques to reduce the number of messages, and techniques to further reduce network contention using task oriented compression. [Bell2008] discusses the performance of different sparse matrix formats on the GPU. There is extensive literature on optimization of Sparse Matrix Vector (SpMV) multiplication, which is closely related to operations on graphs [Xu2010, Guo2010, Oberhuber2010, Feng2011, Heller2012, Koza2012, Pichel2012, Vázquez2012, Maggioni2013]. [Zhong2013] discusses source to source transforms in Medusa to optimize the memory layout by converting from Array of Structures (AoS) to Structure of Arrays (SoA). [Sung2010] presents DL, an approach for automatically transforming data layouts when moving data between the host and the device. [Fang2010] discusses column compression on the GPU and could offer a path to data sharing between the GPU and a graph database.

[Harish2007], [Hong2012], [Merrill2012], [Totem2012], and [Zhong2013] (Medusa) have studied graph traversal on GPUs. Merrill developed the first work efficient implementation of BFS on GPUs, developed adaptive strategies for assigning Threads, Warps, and CTAs to vertices and edges, and optimized frontier expansion using various heuristics to trade off time and space and obtain high throughput for algorithms with dynamic frontiers. Merrill (for BFS) and SYSTAP (for BFS, SSSP, and CC) offer the best results to date for graph processing on a single GPU. [Merrill2012] presents results for up to 4-GPUs in a single workstation on BFS. Merrill stripes the vertices across the GPUs and relies on Unified Virtual Addressing (UVA) for data movement. This approach

did not provide good scaling on most data sets. Gharaibeh (Totem) and Zhong (Medusa) present multi-GPU results for Page Rank (PR) and BFS. Gharaibeh defines a performance model for hybrid CPU/GPU graph processing, tests that model with up to 2 GPUs using random edge cuts, and notes that aggregation can reduce communications costs. Zhong uses 1D partitioning and also tests n-hop partitioning (to increase locality through redundancy). PR does more work per byte and visits all vertices in each round. This allows the simpler frontier and data-dependent parallelism strategies in Medusa and Totem to achieve good scaling. However, algorithms that do less work per byte and which have dynamic frontiers (such as BFS and SSSP) expose this weakness. Both Medusa and Totem have poor scaling on such algorithms. None of the existing approaches shows good scaling to multiple GPUs across a wide range of graphs, algorithms, and data scales. None of the approaches scale to GPU clusters – they are all multiple GPUs in a single node.

[Checconi2012] ranks 1st on the Graph500 on Blue Gene hardware. They use 2D partitioning and a batch parallel "wave" that propagates along the rows of the 2D grid to eliminate redundant vertex updates (and 97% of the communication for those updates). Such batch parallel operations are very efficient for a GPU cluster. BFS is modeled in GAS as a "scatter" operation and thus directly corresponds to the "wave." SSSP is modeled as a gather on the in-edges followed by a scatter on the out-edges. The "wave" could thus provide an efficient basis for the gather and scatter phase of GAS algorithms on GPU clusters.

We believe that good scaling on multiple GPUs and GPU clusters *can* be achieved across a wide range of algorithms, graphs, and data scales by using: a sophisticated partitioning strategy (2D); batch parallel waves that reduce communication costs when accessing vertex state; sophisticated techniques for thread and frontier handling (Merrill); and sophisticated approaches to task scheduling and latency hiding (e.g., Uintah).

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Technical Objectives

Our overall objective is to develop a high-performance, scalable, extensible, easy-to-use, open-source platform for graph analytics on GPUs that is future-proofed against the continuing revolution in many-core hardware. The design will have three primary aspects:

1. A high-level abstraction: The high-level abstraction will make it possible for ordinary programmers to leverage data-parallel evaluation of graph algorithms on GPUs;

2. A data-parallel runtime: An optimized, data-parallel runtime will deliver the potential of GPUs for graph analytics; and

3. A scalable architecture for graph analytics: Large graphs will be automatically decomposed into patches. Operations on large graphs will be decomposed into tasks that operate over those patches. The patches and tasks will be distributed across the resources of a multi-GPU workstation or GPU-enabled compute cluster. A local scheduler on each node will run tasks as their prerequisites are satisfied, optimize the bandwidth utilization of the PCIe bus by intelligent data and task placement, and overlap data movement with computation to hide latency.

3.2 Schedule

The following table provides a schedule of the deliverables in year 1.

Table 1: First Year Deliverables

Month		Milestone	Date	Narrative
1	Oct			
2	Nov			
3	Dec	1	Dec 7, 2012	Software Development Plan
4	Jan	2	Jan 31, 2013	Application Architecture 1
5	Feb			
6	Mar			
7	Apr			
8	May	3	May 20, 2013	Code Level 1
9	Jun			
10	Jul	4	July 9, 2013	Code Level 2
11	Aug			
12	Sep			

3.3 Year 1 Roadmap

During the first program year, we focused on the develop of (a) low-level implementations of graph primitives that extended Merrill's work on BFS; (b) the development of a high-level abstraction (the GAS model); and (c) an efficient data-parallel implementation of that abstraction on a single GPU (MPGraph v2).

The following table provides a high-level summary of the major events in year 1.

Table 2: First Year Summary

Month		Milestone	Narrative
1	Oct		- Contract start
2	Nov		- Kickoff telcon
3	Dec	1	Literature Reviews: - Graph databases (SYSTAP); - Graph processing on GPUs (Davis); - Graph algorithms (Dr. Yan)
4	Jan	2	Identified initial approaches: - Vertex-centric, 2D partitioning (SYSTAP) - Low-level CC algorithm (Davis)
5	Feb		- Import of Merrill's source code (Apache 2.0)
6	Mar		- Exploration of GAS Architecture as basis for scale-out design.
7	Apr		
8	May	3	- GPU GAS POC implemented in Thrust.
9	Jun		- Thrust code evaluated on GPU, Intel Xeon Phi, Intel Thread Building Block, and Open MP. - Approximate BC using GAS API and sampling (SYSTAP); - Exact BC (Davis);
10	Jul	4	- Mid-point review; - Dr. Zhisong Fu starts full-time; - Final review, 2-minute lightning talk; - Initial open source release (MPGraph v1, using Thrust POC).
11	Aug		- Java GAS POC.
12	Sep		- New MPGraph v2 implementation developed by Dr. Fu (10x faster than the original Thrust-based POC).

Some analytics from the summer camp, e.g., approximate Betweenness Centrality, have not been migrated to the newer code base yet. Likewise, there are analytics in our roadmap that we have not yet implemented such as Louvain Modularity, kcore, triangle counting, approximate diameter, graph invariants, etc.

3.4 Future Roadmap

In the future, we planned to focus on multi-GPU scaling. We have outlined several approaches to scaling in the sections above. Our starting point will be 2D partitioning. This will allow us to break down large graphs into patches while minimizing the communications volume. Each patch will be able to fit onto a single GPU. The data movement will be overlapped with the computation in order to maximize the throughput.

3.5 Technical Approach

We are developing an open source, open architecture platform, *MPGraph*[1], for scalable high-performance graph analytics on GPUs. During the first program year, we developed two basic aspects that platform:

- A high-level abstraction (based on the GAS abstraction); and
- A data-parallel runtime (based on Merrill's approach).

We were to begin to tackle the third aspect (scalable analytics) during the 2nd program year.

The version of MPGraph that was evaluated during the Summer Camp was based on a Proof of Concept (POC) using Thrust. This POC could outperform the CPU for many graphs, but the code did not use any of Merrill's concepts for efficient data-parallel operations on graphs. During the summer camp we implemented new algorithms over the Thrust-based GAS Engine, including CC, and an approximate version of Betweenness Centrality (BC) based on edge-sampling [Brandes2001]. Sampling not only allowed us to scale to larger data sets, but also dramatically accelerated BC, which is known to scale poorly as a function of the size of the graph (in fact, the exact version of BC ran for a week without completing on the target data set). Sampling can also speedup graph algorithms that run against indices, dramatically reducing the IOs required for a computation. We also collaborated with the Johns Hopkins team to demonstrate a fast and accurate approach to triangle counting based on the cube of the eigenvalues.

Since the Summer Camp we have developed a completely new code base that is significantly faster than the original POC – see **Figure 2**. We developed a BFS reference implementation for GraphLab. BFS is implemented as a Scatter. The speedups for SSSP are in reference to GraphLab's SSSP implementation, which also does all the work in the Scatter phase.

[1] *M*assively *Parallel* Graph, https://sourceforge.net/projects/mpgraph/

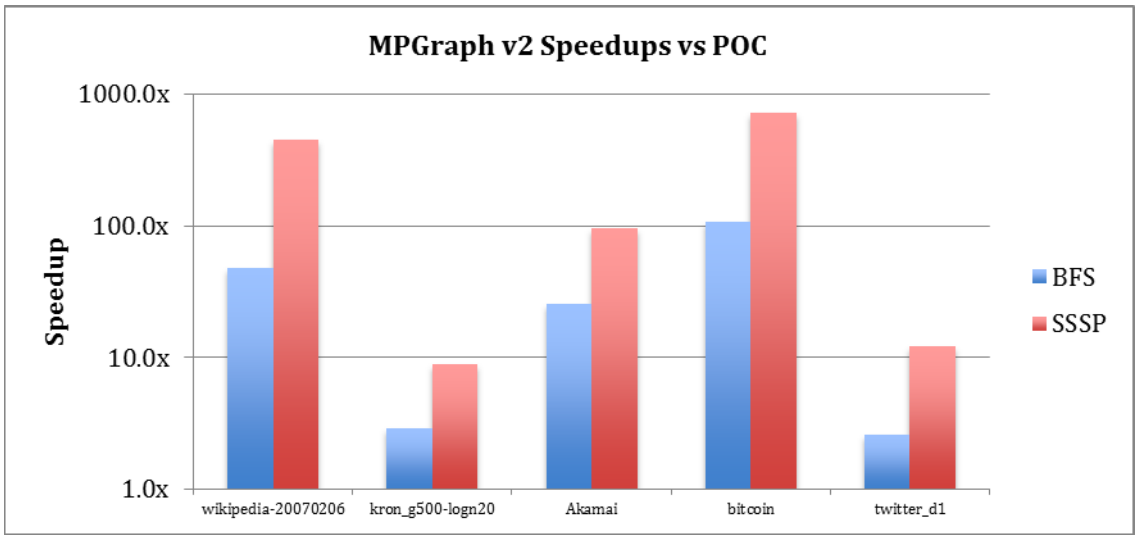


Figure 2: MPGraph v2 Speedups versus POC

The new code base will be released as MPGraph *v2*. This code base utilizes Merrill's techniques for efficient data-parallel operations on graphs and encapsulates them within the GAS abstraction layer. In particular, we use:

- A compact frontier queue;
- Dynamic granularity to assign threads, warps, and CTAs to edges and vertices; and
- Bitmask cull, history cull, and warp cull heuristics to remove duplicate vertices from the frontier.

Detailed performance results for MPGraph are given in the Analysis section below.

Finally, we have developed a Java-based implementation of the GAS abstraction. This implementation can run against in-memory data structures or disk-based indices. We were to use the Java implementation to prototype the target design for the GPU GAS Engine, explore approaches to deal with attributed graphs, and examine options to target the GPU from languages other than CUDA (e.g., Java / Scale, Python, etc.).

3.6 Open Source Project

Our goal is to develop a high-performance and scalable open source implementation offering an open architecture and a high-level abstraction. Our efforts to date have established a data-parallel backend and demonstrated that the data-parallel backend can be encapsulated within a high-level abstraction. We are now working to extend the data-parallel backend to other algorithms, improve the encapsulation of the abstraction, and lay the ground work for a robust, open source, open architecture platform. In

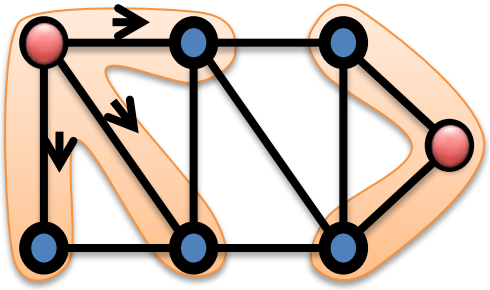
addition, we have been working to establish performance regression tests in the NVIDIA performance lab.

Our first open source release was on July 30th, 2013. We had been planning a second release in December, 2014. The second release would provide an order of magnitude performance increase for common graph analytics and will make it easier for people to write new algorithms.

3.7 High-Performance Graph Analytics

Since the Summer Camp, we have implemented four high-performance graph analytics for the GPU using the GAS abstraction: BFS, SSSP, CC, and PR. These analytics are summarized in **Table** 3. The backend for these analytics uses a compact frontier queue, dynamic granularity, and a variety of cull techniques to obtain high performance. We present results for BFS and SSSP below.

The GAS API presents a vertex-centric abstraction [Pregel2010]. The Gather operation may read on in-edges, the out-edges, or both. The Apply operation acts on vertices in the current frontier. The Scatter operation distributes messages to other vertices, and again may operate on the in-edges, the out-edges, or both. A large number of interesting algorithms can be implemented within this abstraction and then scaled transparently to run on CPU clusters.



Gather: collect information about my neighborhood
Apply: update my value
Scatter: signal adjacent vertices

Figure 3: GAS Graphic

BFS is modeled in GAS as a Scatter operation. The GAS implementation of BFS therefore allows us to directly measure the abstraction overhead for that algorithm. The GAS implementation of SSSP is a Gather plus a Scatter[2]. SSSP is thus the simplest algorithmic extension of BFS. Details on the GAS algorithms are presented in **Table** 3.

Both BFS and SSSP use a compact frontier. The frontier is realized using Merrill's work-efficient approach. It provides a compact queue of vertices to visit in the next iteration. Since many vertices in a given iteration can have the same neighbor, the expansion of the frontier generally produces a large number of duplicates. However, filtering duplicates globally across the GPU is a relatively expensive operation. Therefore, the code uses low-cost heuristics to eliminate many, but not all, of the duplicate vertices.

[2] GraphLab actually breaks their own abstraction for SSSP as explained in depth below.

Duplicate vertices that remain in the frontier can cause duplicate work in the next iteration. The compact frontier design represents a tradeoff between time and space. Higher performance is obtained with the compact frontier, but the frontier queue consumes a significant portion of the memory on the GPU.

The Gather and Scatter phases of the GAS abstraction are modeled using *expand* and *contract* operations in GPU kernels. The edge-frontier is the set of edges to be traversed in a given iteration, based on the current vertex-frontier. In practice the edge-frontier is modeled as the vertices that are accessible by traversing those edges. The edge-frontier will contain duplicate vertices if the same vertex is discovered by traversing more than one edge in a given iteration. Such simultaneous discovery is quite common and eliminating duplicate vertices efficiently is a significant challenge for the GPU. The vertex-frontier is the distinct subset of vertices in the previous edge-frontier that will be visited in the next iteration.

The *expand* kernel generates an edge-frontier from the current set of vertices by traversing the edges for the vertices in the current vertex-frontier. The *contract* kernel then reduces the edge-frontier to a compact vertex-frontier for the next iteration. Contraction is concerned with status lookup (resolving whether the vertex should be updated or scheduled) and eliminating duplicate work from the new vertex-frontier using heuristics to cull duplicate vertices. In practice, the expand and contract operations may be realized as separate kernels (two-phase) or combined into a single kernel (either expand-contract or contract-expand). Merrill showed that the two-phase kernel performed better if the vertex-frontier was large, but that the combined kernels provided better throughput for smaller frontiers such as roadmaps or the Bitcoin data set. In our work, we have focused on two-phase kernels, but it is possible that higher performance could be realized for Bitcoin and similar data sets by developing a combined kernel and then using either the combined kernel or the two-phase kernel as appropriate for the data set, e.g., a hybrid kernel.

Unlike BFS or SSSP, the CC and PR algorithms visit all vertices in the initial round and may visit a substantial proportion of all vertices in every round. We are currently experimenting with a variety of techniques designed to optimize the frontier for such workloads.

In general, graph algorithms are memory bound because they (a) perform very little work per byte; and (b) have poor locality. PR does more work per byte than the other algorithms that we have explored. This makes it a good candidate for GPU acceleration, and other researchers have already demonstrated that it is possible to obtain good speedups on PR on a GPU [Totem2012] and on multiple GPUs [Duong2012, Zhong2013]. Obtaining good performance and good multi-GPU speedups for BFS, SSSP, and similar fundamental algorithms is much harder because these algorithms put significantly more demand on the memory systems.

Table 3: Graph Analytics Analyzed in This Report

Label	Name	Description	GAS Approach
BFS	Breadth First Search	Label all vertices based on their minimum distance (hops) from a given starting vertex. BFS is a fundamental building block for many graph algorithms.	- Schedule starting vertex. - Gather: NOP - Apply: if not visited, set level to round#. - Scatter iff visited first time.
SSSP	Single Source Shortest Path	Find the shortest path between a given vertices and all directly or indirectly connected vertices in the graph such that the sum of the weighted edges is minimized.	- Gather: min over in-edges [3]. - Apply: value = min(self, gather). - Scatter: out-edges iff value changed.
CC	Connected Components	Find the distinct and non-overlapping subgraphs of a graph. Each such subset consists of vertices that are connected by a path.	- Visit all vertices in the first round. - Gather: min of source vertex label over all edges.[4] - Apply: value = min(self, gather). - Scatter: all-edges iff value changed.
PR	Page Rank	Page rank assigns weights to the vertices in a graph based by on the relative "importance" as determined by the patterns of directed links in the graph.	- Visit all vertices in the first round. - Gather: sum of (neighbor_value / neighbor_num_out_edges) over the in-edges. - Apply: value = 0.15 + (1.0 – 0.15) * gather - Scatter: iff value has significantly changed (fabs(old-new) > *epsilon*) where epsilon controls the degree of convergence before the algorithm terminates.[5]

[3] The GraphLab implementation of SSSP (and CC) does not use a Gather phase. Instead, it breaks the GAS abstraction and communicates the new path lengths as messages during the Scatter phase. This means that it traverses ½ of the number of edges of the "pure" GAS implementation. We have implemented this optimization in MPGraph v2 and are examining ways of extending the GAS abstraction that will allow us to capture the more efficient algorithm in an easy to express program.

[4] The GraphLab implementation of CC breaks the GAS abstraction in exactly the same manner as SSSP.

[5] In fact, page rank is typically run to a fixed number of iterations in order to remove side-effects from the order in which floating point updates are computed. This produces more reliable convergence.

3.8 Evaluation

Merrill used two different mechanisms to estimate the speedup offered by a GPU over a CPU for graph processing. First, he directly compared with the best published results for multi-core CPU algorithms [Leiserson2010, Agarwal2010]. Both studies reported sub-linear scaling per CPU core and Merrill quotes harmonic means speedups of 8.1x and 4.2x versus their 4-core and 8-core parallel BFS results. Second, Merrill implements a single-core version of BFS, verifies the performance of that implementation against published single-core results, and then uses idealized linear scaling to identify an upper bound for multi-core performance. Using this idealized linear scaling assumption, he estimated the GPU speedup at 12x – this is the point at which the GPU would be faster than the CPU for ½ of the measured data sets.

As Merrill noted, comparing CPU and GPU performance is challenging, in part because good CPU implementation of graph algorithms are difficult to write. For example, [Zhong2013] develops CPU implementations for BFS, SSSP, and PR using MTGL in order to characterize the performance of the GPU implementation, but those CPU implementations have poor scaling. Rather than attempt to develop best of breed CPU implementations for each algorithm, we have focused on a comparison with a widely deployed graph mining platform, GraphLab.

Our goal has been to compare the performance of the GPU with the CPU within the context of a high-level abstraction for graph algorithms. To do this, we measure actual speedups for BFS and SSSP as implemented in the current development version of MPGraph [6] against GraphLab (v2.2). GraphLab is a broadly deployed CPU-based graph mining platform that uses the same vertex-centric abstraction layer.

For our GPU evaluations we used a NVIDIA c2075 (Fermi architecture) [7].

The CPU platform was a machine containing a 3.33 GHz X5680 CPU chipset. This is a dual-socket Westmere chipset that contains 12 physical cores and 12 MB of cache. The machine contains 24 GB of 1333 MHz ECC memory. The software environment is RedHat 6.2 Beta. CPU code was compiled with gcc (GCC) 4.4.6 20110731 (Red Hat 4.4.6-3). The results were obtained using the synchronous engine for GraphLab due to core faults with some data sets when using the *asynchronous* engine [8].

---

[6] https://sourceforge.net/projects/mpgraph/

[7] We have encountered some memory access problems with larger graphs on the K20c. We have been able to replicate the problem in Merrill's original source code and are working to resolve this issue. This memory access problems manifests commonly for large graphs on the K20, but is rarely observed for the Fermi architecture cards. Therefore we have used the c2075 for the larger graphs in this study.

[8] [Pearce2010] has shown improved scaling for CPU architectures using asynchronous processing for BFS, SSSP, and CC. GraphLab v2.2 does include an asynchronous engine, but the asynchronous engine was not used due to problems (segment faults). However, tests by the GraphLab developers (private communication) with PR on the kron_g500_logn20 data set failed to demonstrate a speedup over the synchronous engine. For 8 CPU cores, the run times were 10.7s for sync versus 18.5s for async; 33% fewer updates were performed, but the throughput was slower. The GraphLab asynchronous engine should be re-evaluated in the future.

For each data set, we pre-selected 100 vertices at random and filtered the vertices to make sure that each selected vertex had at least one out-edge. For both BFS and SSSP, we then performed 100 trials, each trial using a different starting vertex. For GraphLab, we then varied the number of CPU cores that were used by the platform and report results for 1, 2, 4, 8, 12, 16, and 24 cores. The machine has 12 physical cores. Results reported for more than 12 CPU cores use hyper threading.

3.9 Data Sets

We present results for the following data sets. Two of these data sets were used by Merrill in his analysis (wikipedia-20070206 and kron_g500-logn20). This gives us a basis for direct comparison with Merrill's results. Three of the data sets are from the XDATA Summer Camp activity.

Table 4: Data Sets

data set	#vertices	#edges	source
wikipedia-20070206	3,566,907	45,030,389	http://www.cise.ufl.edu/research/sparse/MM/Gleich/wikipedia-20070206.tar.gz
kron_g500-logn20	1,048,576	89,239,674	http://www.cise.ufl.edu/research/sparse/MM/DIMACS10/kron_g500-logn20.tar.gz
Akamai	24,339,217	25,602,011	DARPA
Bitcoin	6,297,539	28,143,065	DARPA
twitter_d1	1,851,583	17,197,688	DARPA

A few points are worth calling out about these data sets:

- The kron_g500-logn20 data is a scale-free graph.
- The Bitcoin has a small frontier and a very long tail. For bitcoin, only a very few vertices were active in any given iteration and a large number of iterations were required for BFS and SSSP (on the order of 8000 iterations).
- The Akamai, Bitcoin, and twitter_d1 data sets are Challenge Problem data sets from the XDATA Summer Camp activity.

4.0 RESULTS AND DISCUSSION

4.1 Improvements Since the Summer Camp

Figure 4 shows speedups of MPGraph v2 and the initial POC implementation, which used Thrust, over the performance of GraphLab using a single CPU core. The POC provided speedups for BFS, but was actually slower than the CPU for SSSP on many data sets (wikipedia, kron_g500logn20, and Bitcoin).

Our first action after the summer camp was to review the POC implementation and determine whether it could be optimized. We attempted several optimizations, including eliminating unnecessary work in the gather and scatter phases and generating a compact frontier. These optimizations resulted in only very minor performance improvements (on the order of 10% or less).

In order to obtain a significant performance improvement, we replaced the Thrust-based POC. The original POC relied on Thrust to assign threads to vertices and edges, rather than using the data-dependent adaptive techniques developed by Merrill. In addition, the original POC modeled the frontier with a Boolean flag on each vertex. This approach works well enough for small graphs, but Merrill's approach using a compact frontier and adaptive techniques to assign threads, warps, and CTAs to edges and vertices is much more efficient for larger graphs, for graphs with small frontiers and long tails, such as Bitcoin, and for the irregular access patterns associated with BFS or SSSP – as seen in **Figure** 4

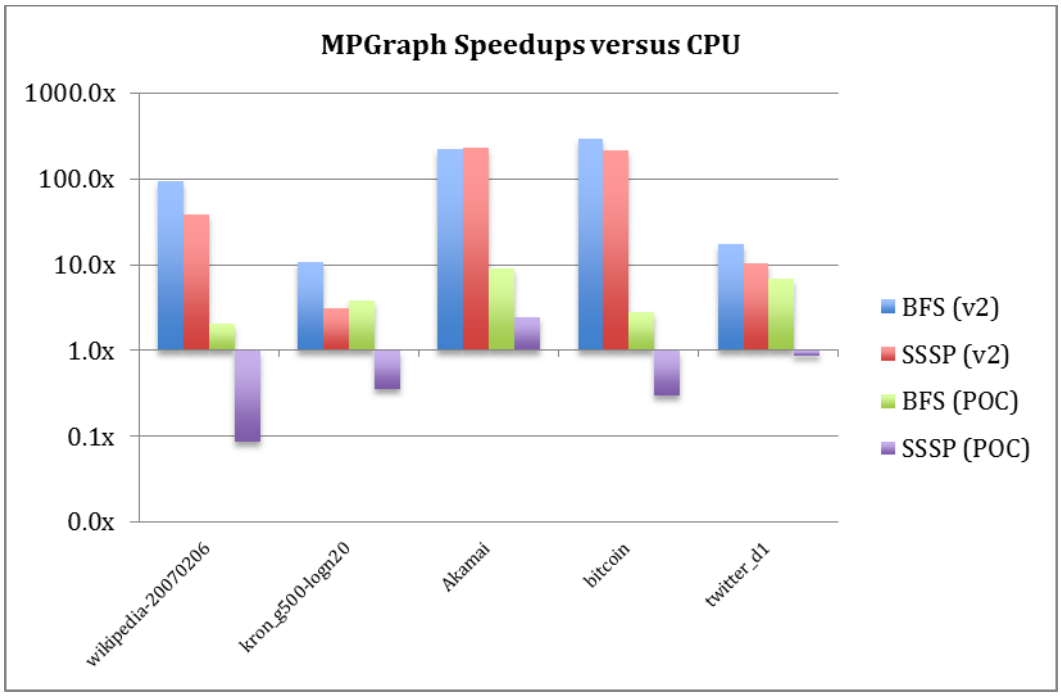


Figure 4: MPGraph Speedups versus CPU

4.2 BFS

Figure 5 depicts speedups as a function of the number of CPUs cores for BFS, and the speedup of the GPU as a function of a single CPU core. The measured CPU speedups for BFS are in fact much lower than those assumed by Merrill. Some things of interest:

- Overall, GraphLab shows very little scaling as we increase the number of CPU cores. For some data sets (kron_g500-logn20 and twitter_d1), CPU performance actually decreases as we add more CPU cores. For the other data sets, adding more cores does not produce any significant speedup. For all data sets, the actual speedup or slowdown of GraphLab as a function of CPU cores is very limited. The maximum speedup is 3.1x (Akamai with 24 cores). The maximum slowdown is 0.2x (kron_g500-logn20 with 16 cores).
- Speedups decrease as a function of the data scale. For example, GraphLab has a speedup of 8.65x for 24 cores over its single core performance for the delanuey_n13 data set (not shown), with only 8,192 vertices and 24,547 edges. This suggests that the CPU performance declines quickly (a) once the data no longer fit into the CPU cache; and (b) as the CPU memory bandwidth becomes saturated.
- There is an interesting, and unexplained, boost from hyper threading when all 24 cores are in use. One hypothesis is that hyper threading allows the CPU to have more memory requests in flight and allows the machine hide more of the latency associated with memory fetches.
- GraphLab is not competitive with the best CPU BFS implementations. For example, [Leiserson2010] shows a 6x scaling on 8 cores for the wikipedia-20070206. For the same data set, GraphLab scales by 0.88x on 16 cores and 1.44x on 24 cores (the hyper threading boost). We believe that this reflects bias in the development of GraphLab towards a general purpose platform. However, it is important to maximize the per-node throughput for distributed systems in order to scale economically.

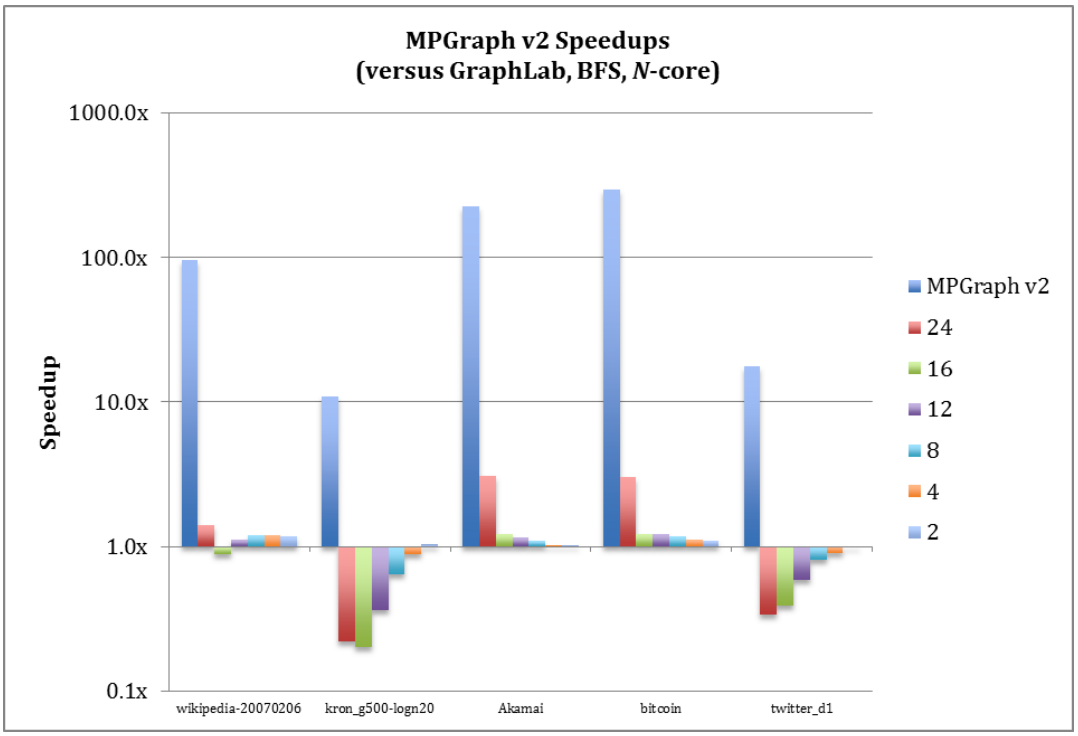


Figure 5: MPGraph speedups over the CPU (BFS)

4.3 SSSP

As seen in **Figure** 6, the performance of MPGraph for SSSP is broadly consistent with the results for BFS as seen in Figure 4. One of the challenges for the GPU implementation was to efficiently filter the remote vertices before adding them to the frontier in order to minimize the size of the frontier and the amount of duplicate work performed. Conceptually, SSSP is very similar to BFS. However, SSSP can revisit vertices that have already been labeled when they are discovered through a longer traversal (more hops) with less total weighted path length. Because of this certain optimizations that are possible for BFS cannot be performed for SSSP. For example, Merrill's BFS code does not insert remote vertices into the frontier if they have already been visited. GraphLab performs a similar optimization, filtering the vertices before adding them to the new frontier based on whether the new value for the remote vertex could be reduced by an update pushed by the local vertex along the edge to the remote vertex. This decision can be made without considering the edge weight for BFS. For SSSP, we need to consider the weight of the source vertex, the edge weight, and the current value of the remote vertex to make the same decision. This introduced a new non-coalesced access pattern into the code, which significantly decreased the throughput of SSSP on the GPU when compared to BFS. We fixed this performance problem by maintaining the vertex state in a compact vector, which provides coalesced access.

Another challenge with SSSP, is computing the minimum over the path lengths for the new frontier. In our GPU implementation, the expansion step accesses the out-edges

and generates a new frontier from old frontier. We also keep a predecessor value array (so named because it captures the predecessors of the current frontier in a level synchronous BFS traversal) to maintain the values of the predecessors of the vertices in the new frontier after expansion. In the contraction step, the minimum value for the vertices of the new frontier is computed with atomicMin. Atomic operations are relatively expensive for Fermi and higher performance is anticipated for Kepler [9].

Both MPGraph v2 and GraphLab v2.2 implement an optimization for SSSP that eliminates the Gather phase of the algorithm. Instead of collecting the new path lengths in a Gather phase, the Scatter phase breaks the encapsulation of the GAS API to transmit the new path lengths to the remote vertex. This optimization eliminates roughly half of the edge traversals [10]. GraphLab performs slightly less well for SSSP when compared to its own performance on BFS. We attribute this to the messaging protocol used by their SSSP implementation, which appears to be slightly less efficient rather than their protocol for normal Scatter operations, such as BFS. This leads to smaller speedups (the speedups for Akamai and Bitcoin are nearly 1.0x for GraphLab) and larger slowdowns (for kron_g500logn20 and twitter_d1).

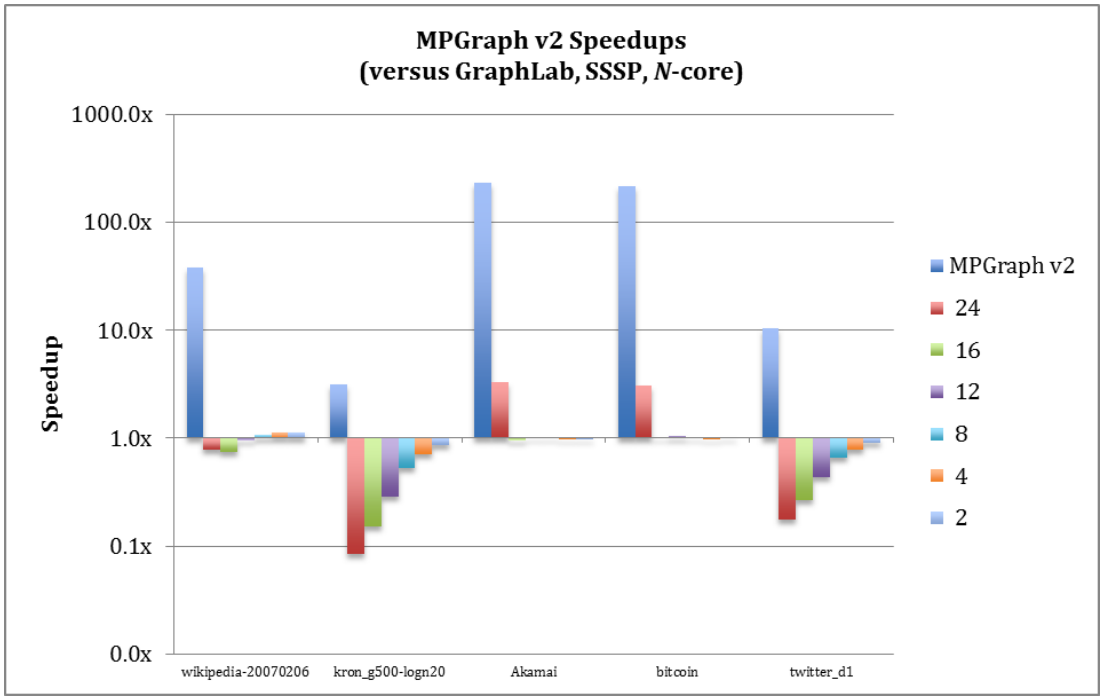


Figure 6: MPGraph speedups over the CPU (SSSP)

[9] We did not collect data on the Kepler K20 cards due to a memory access problem in the library that was aggravated by the K20. The reported numbers are for an NVIDIA Fermi c2075.

[10] The Thrust-based POC did not implement this optimization for SSSP. This is why its SSSP performance was significantly worse when compared to its own BFS performance.

4.4 CC/PR

We are currently developing GAS implementations for the GPU for CC and PR. The GAS algorithms themselves are easy to express. However, these algorithms have different characteristics from both BFS and SSSP. Specifically, CC and PR tend to have larger frontiers, especially in the first several iterations. PR does more work per byte, which tends to favor the GPU. However, PR requires us to compute the sum of its neighbors. This will produce the wrong result if there are duplicates in the frontier. Thus, for PR, it is necessary eliminate all duplicates from the vertex-frontier rather than relying on cull heuristics.

Early results suggest a 10x speedup over the thrust based POC from the summer camp, but we need to examine the performance of the data-parallel backend in more depth before reporting results on these algorithms. We would also like to compare with [Zhong2013] (Medusa) and [Totem2012], both of which have results for PR on the GPU.

The UC Davis CC implementation described in their report is based on a GPU algorithm for graph connectivity [Soman2010] that has intrinsically better performance than CC expressed as a GAS program (see **Table** 3). The goal of a high-level abstraction is to make easy to capture a wide range of algorithms. However, the high-level abstraction itself is a work in progress. Future research will explore whether algorithms such as [Soman2010] can be captured through extensions to the GAS API. Other potential extensions to the high-level abstraction would expose mechanism for sampling, leveraging sketches, etc.

4.5 Cost per GTEPS

It is difficult to characterize the price performance for CPU and GPU solutions. CPU solutions exist today, but their throughput is remarkably poor in comparison to the best known algorithms for BFS, SSSP, or CC. GPU solutions require an investment in basic research to develop solutions that deliver the high-performance potential of the GPU, encapsulate the capabilities of the hardware within a high-level abstraction, and can scale to multiple GPUs and GPU clusters.

Assuming that we purchase commodity hardware solutions, a 4-core CPU server might cost $4000 and deliver (as a very conservative estimate) $1/10^{th}$ of the throughput of a single GPU. Adding more CPUs does not appear to be a cost effective technique for obtaining higher performance. CPU clusters increase the data scale of the problems that can be addressed and provide (sub-linear) scaling in throughput.

If we install a single Kepler GK110 GPU into the same $4000 server, we will raise its price by between $650 (for the GTX780 card), $1000 (for the Titan), and $3000 (for the K20). At the same time, we will have 10x more throughput. Thus, we have increased the $/GTEPS by between 5x (10x faster, less than twice the price) and 8x (10x faster, with only 16% more for the GTX780). These are very conservative estimates. The actual price/performance ratio is likely to be significantly better as measured against

existing graph mining solutions. More realistic speedups appear to be on the order of 50x for a single GPU when comparing with solutions such as GraphLab.

GPU-based solutions appear to be a clear winner for graphs with up to 1 billion edges. Beyond that point, we need more research to understand how a GPU-based solution will scale to multiple GPUs and GPU clusters.

5.0 CONCLUSIONS

Efficient graph algorithms are challenging to write, regardless of the hardware architecture. High-level abstractions are necessary in order to expose the potential performance of the underlying hardware architecture to everyday users. In this effort, we have demonstrated that an efficient data-parallel GPU implementation of low-level primitives can be encapsulated to create a high-performance library for writing graph algorithms. The underlying data-parallel primitives exploit a variety of adaptive strategies and deliver high performance across a wide range of graphs. Our experience has shown us that new adaptive strategies may need to be developed to handle graph algorithms where the frontier is the vast majority of the vertices in the graph, for example, CC and PR. Once developed, those adaptive strategies can be encapsulated within the same high-level abstraction and disappear from the concern of the graph analytic developer.

Graph algorithms are known to be memory bound. The GPU architecture has ten times the memory bandwidth of the CPU and efficient GPU graph traversal implementations are able to deliver significant speedups in comparison with CPU architectures. Merrill assumed idealized linear scaling for CPU graph algorithms. However, empirical evaluation has shown that, for at least one widely used platform, actual CPU scaling is much worse than linear.

[Ediger2013] demonstrated linear scaling on the XMT using relatively slow clocks and a low-latency interconnect. [Checconi2012] has shown good scaling to extremely large graph on BFS using Blue Gene hardware. In contrast, GPUs represent a cost affordable approach to high performance graph analytics on commodity hardware. Existing work has shown good scaling to multiple GPUs in a single workstation on Page Rank, but not on Breadth First Search – the fundamental building block of graph algorithms. More research is required in order to realize the potential of GPUs as a commodity platform for high performance graph processing.

The GAS abstraction makes it easy to express a large variety of graph algorithms. However, there are some algorithms, such as connected components, where the best known algorithms cannot be easily expressed as a GAS program [Soman2010]. Further research is required to introduce more flexibility into the high-level abstraction without either (a) limiting the ability to execute algorithms in an efficient, data-parallel fashion; or (b) introducing features into the architecture that would prevent scaling to multiple GPUs or GPU clusters. Some possibilities include abstractions for sampling the edges of the graph [Brandes2001, Tsourakakis2009, Haim2010], computations on

sketches [Ahn2012], approximations on streaming graphs [Kutzkov2013], managing mutations on the graph (cutting edges or removing vertices), introducing explicit temporal dimensions into the graph, or maintaining auxiliary data structures, such as probabilistic frequent item sets [Khan2010].

5.1 Next Steps

Unfortunately this effort was shortened due to an early termination. During the coming year, we had planned to complete the current refactoring and then explore approaches to scale computations to multiple GPUs. We view multi-GPU support as the critical next step to enabling a new low-cost, high-performance capability for graph analytics. Our goal was to achieve good scaling without sacrificing the ease of use introduced by the high-level abstraction. We believe that success for the platform lies in rapidly developing a high-performance and scalable implementation with an open architecture and a high-level abstraction. This would allow other researchers to experiment with and extend the framework and will support integration into non-CUDA environments such as Java, python, and R.

5.2 Future Work

Our goal has been to develop a scalable, high-performance, open-source, and open architecture platform for GPU based graph analytics. We will continue to pursue this and believe that the experimental results presented in this report not only confirm the capability of the GPU as the basis of a high-performance graph analysis platform, but also draw new attention to the severe throughput limitations of existing CPU based graph analysis programs and provide evidence for the importance of GPU-based solutions as a low-cost path to high performance on graphs.

Our intention was to shift the focus of development from (a) single GPU performance and (b) the high-level abstraction to the development and evaluation of a single-machine, multi-GPU architecture.

The following list outlines some of our planned research topics for the development of a scalable data-parallel graph analytics platform:

- Scaling to GPU clusters through a collaborative research effort with Dr. Berzins and the University of Utah Scientific Computing and Imaging Institute;
- Optimized memory layouts and compression techniques;
- Data-parallel operations on per-edge vectors and sets. The current code assigns one thread to each edge and vertex. This is appropriate when the edge or vertex state is a scalar, but does not provide sufficient parallelism when the edge state is a vector or set;
- Exposing the high-level abstraction to other programming languages (Java/Scala, Python, R, etc.);
- Improved robustness for the open source platform; and

- Improved outreach for the open source platform.

The following list outlines some research topics that would go beyond the basic capabilities to develop a more novel platform:

- Fast algorithms for graph pattern matching and approximate graph pattern matching. This family of techniques is relevant where there is an *a priori* model to be matched in the data;
- Fast algorithms for graph pattern mining and aggregation, e.g., [Khan2010]. This family of techniques provides tools for bottom-up analysis to identify the top-k interesting patterns;
- Fast approximation techniques, including sampling [Brandes2001, Tsourakakis2009, Haim2010] and sketches [Ahn2012], and means for characterizing the estimation error in those approximations; and
- Streaming graphs, e.g., [Kutzkov2013].

6.0 REFERENCES

[Ahn2012] K. Ahn, S. Guha, and A. McGregor. "**Graph sketches: sparsification, spanners, and subgraphs.**" In *Proceedings of the 31st symposium on Principles of Database Systems* (PODS '12), Markus Krötzsch (Ed.). ACM, New York, NY, USA, 5-14, 2012.

[Agarwal2010] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader. **"Scalable graph exploration on multicore processors."** In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-11. IEEE Computer Society, 2010.

[Bader2006] D. Bader and K. Madduri, "**Designing Multithreaded Algorithms for Breadth-First Search and *st*-connectivity on the Cray MTA-2.**" *Proc. The 35th International Conference on Parallel Processing (ICPP)*, Columbus, OH, August 2006.

[Bader2008] D. Bader and K. Madduri, **"Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks."** in IPDPS, pp. 1–12., 2008.

[BEAMER2012] S. Beamer, K. Asanovic, and D. Patterson. "**Direction-optimizing breadth-first search.**" In High Performance Computing, Networking, Storage and Analysis (SC), pp. 1-10. IEEE, 2012.

[Bell2008] N. Bell and M. Garland, **"Efficient Sparse Matrix-Vector Multiplication on CUDA."** in *NVIDIA Technical Report NVR-2008-004*, December 2008.

[Berry2007] J. Berry, B. Hendrickson, S. Kahan, P. Konecny, **"Software and Algorithms for Graph Queries on Multithreaded Architectures."** IPDPS, pp.495, 2007 IEEE International Parallel and Distributed Processing Symposium, 2007.

[BONCZ1999] P. A. Boncz, S. Manegold, and M. L. Kersten. "**Database Architecture Optimized for the New Bottleneck: Memory Access"**. In VLDB, 1999.

[Brandes2001] U. Brandes. "**A faster algorithm for betweenness centrality**." *Journal of Mathematical Sociology* 25, no. 2 (2001): 163-177.

[Buluc2011] A. Buluç, and J. Gilbert. **"The Combinatorial BLAS: Design, implementation, and applications."** International Journal of High Performance Computing Applications 25, no. 4 (2011): 496-509.

[Catalyurek1999] U. Catalyurek and C. Aykanat. **"Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication."** IEEE Trans. Parallel Dist. Systems, 10(7):673–693, 1999.

[Catalyurek1999-patoh] U. V. Catalyurek and C. Aykanat. **"PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0."**, Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey, 1999.

[CCGRID13] J. Schmidt, M. Berzins, J. Thornock, T. Saad, J. Sutherland. **"Large Scale Parallel Solution of Incompressible Flow Problems using Uintah and hypre,"** SCI

Technical Report, No. UUSCI-2012-002, *SCI Institute, University of Utah*, 2012. Proccedings of CCGRID 13 Conference Delft Holland, IEEE.

[Checconi2012] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. Choudhury, and Y. Sabharwal. "**Breaking the speed and scalability barriers for graph exploration on distributed-memory machines."** In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pp. 1-12. IEEE, 2012.

[Chen2012] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. "**Improving large graph processing on partitioned graphs in the cloud."** In Proc. of the Third ACM Symposium on Cloud Computing (SoCC '12), 2012.

[Concurrency2012] Q. Meng, M. Berzins. **"Scalable Large-scale Fluid-structure Interaction Solvers in the Uintah Framework via Hybrid Task-based Parallelism Algorithms,"** SCI Technical Report, No. UUSCI-2012-004, *SCI Institute, University of Utah*, 2012. Accepted by Concurrency and Computation. (To appear)

[Devine2002] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. "**Zoltan data management services for parallel dynamic applications**." *Computing in Science & Engineering* 4, no. 2 (2002): 90-96.

[Devine2006] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek. "**Parallel hypergraph partitioning for scientific computing**." In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10-pp. IEEE, 2006.

[Duong2012] N. Duong, Q. Nguyen, A. Nguyen, and H. Nguyen. "Parallel PageRank computation using GPUs." In *Proceedings of the Third Symposium on Information and Communication Technology*, pp. 223-230. ACM, 2012.

[Ediger2013] D. Ediger and D. Bader, "Investigating Graph Algorithms in the BSP Model on the Cray XMT." 7th Workshop on Multithreaded Architectures and Applications (MTAAP), Boston, MA, May 24, 2013.

[EUROPAR] M. Berzins, Q. Meng, J. Schmidt, J.C. Sutherland. **"DAG-Based Software Frameworks for PDEs,"** In *Proceedings of Euro-Par 2011 Workshops, Part I*, Lecture Notes in Computer Science (LNCS) 7155, Springer-Verlag Berlin Heidelberg, pp. 324--333. August, 2012.

[Fang2010] W. Fang, B. He, and Q. Luo. "**Database compression on graphics processors.**" *Proceedings of the VLDB Endowment* 3, no. 1-2 (2010): 670-680.

[Feng2011] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, and Z. Shao. "**Optimization of sparse matrix-vector multiplication with variant csr on gpus.**" In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pp. 165-172. IEEE, 2011.

[GraphChi2012] A. Kyrola, G. Blelloch, and C Guestrin. "**GraphChi: Large-scale graph computation on just a PC.**" In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 31-46. 2012.

[GraphLab2010] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. "**Graphlab: A new framework for parallel machine learning.**" *arXiv preprint arXiv:1006.4990* (2010).

[GraphLab2012] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein. "**Distributed GraphLab: A framework for machine learning and data mining in the cloud.**" *Proceedings of the VLDB Endowment* 5, no. 8 (2012): 716-727.

[GraphLab2013] Y. Low. "**GraphLab: A Distributed Abstraction for Large Scale Machine Learning**." PhD thesis., University of California, 2013.

[Guo2010] P. Guo, and L. Wang. "**Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus.**" In *Computational and Information Sciences (ICCIS), 2010 International Conference on*, pp. 1154-1157. IEEE, 2010.

[Haim2010] A. Haim. "**Counting triangles in large graphs using randomized matrix trace estimation.**" *Proceedings of KDD-LDMTA* 10, 2010.

[Harish2007] P. Harish and P. Narayanan. "**Accelerating large graph algorithms on the GPU using CUDA.**" In *High performance computing–HiPC 2007*, pp. 197-208. Springer Berlin Heidelberg, 2007.

[He2008] H. He and A. K. Singh. **"Graphs-at-a-time: Query language and access methods for graph databases."** In SIGMOD, pages 405–418, 2008.

[Heller2012] M. Heller, and T. Oberhuber. "**Adaptive Row-grouped CSR Format for Storing of Sparse Matrices on GPU.**" *arXiv preprint arXiv:1203.5737* (2012).

[Hong2012] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. "**Green-Marl: a DSL for easy and efficient graph analysis**." In *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 349-362. ACM, 2012.

[IPDPS10] J. Luitjens, M. Berzins. **"Improving the Performance of Uintah: A Large-Scale Adaptive Meshing Computational Framework,"** In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS10)*, Atlanta, GA, pp. 1--10.

[Kaldeway2012] T. Kaldewey, et al. "**GPU join processing revisited.**" Proceedings of the Eighth International Workshop on Data Management on New Hardware. ACM, 2012.

[Karypis1995] G. Karypis and V. Kumar. "**A fast and high quality multilevel scheme for partitioning irregular graphs.**" International Conference on Parallel Processing, pp. 113-122, 1995.

[Karypis1997] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. **"Multilevel hypergraph partitioning: application in VLSI domain."** In Proc. 34th Conf. Design Automation, pages 526 – 529. ACM, 1997.

[Karypis1999] G. Karypis and V. Kumar, "**Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs**", *Siam Review* 41, no. 2 (1999): 278-300.

[Khan2010] A. Khan, X. Yan, and K. Wu. "**Towards proximity pattern mining in large graphs.**" In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 867-878. ACM, 2010.

[Koza2012] Z. Koza, M. Matyka, S. Szkoda, and Ł. Mirosław. "**Compressed multiple-row storage format.**" *arXiv preprint arXiv:1203.2946* (2012).

[Kutzkov2013] K. Kutzkov and R. Pagh. "**On the streaming complexity of computing local clustering coefficients.**" In *Proceedings of the sixth ACM international conference on Web search and data mining* (WSDM '13). ACM, New York, NY, USA, 677-686, 2013.

[Leiserson2010] C. Leiserson and T. Schardl. "**A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers).**" In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pp. 303-314. ACM, 2010.

[Lou2010] L. Luo, M. Wong, and W. Hwu. **"An effective GPU implementation of breadth-first search."** In *Proceedings of the 47th Design Automation Conference*, pp. 52-55. ACM, 2010.

[Lugo2012] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. **"A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis."** In SDM, vol. 12, pp. 930-941. 2012.

[Maggioni2013] M. Maggioni, and T. Berger-Wolf. "**An Architecture-Aware Technique for Optimizing Sparse Matrix-Vector Multiplication on GPUs.**" *Procedia Computer Science* 18 (2013): 329-338.

[Merrill2012] D. Merrill, M. Garland, and A Grimshaw. "**Scalable GPU graph traversal**." In *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 117-128. ACM, 2012.

[Nebo2012] C. Earl, J. Sutherland, M. Might. **"Nebo: A Domain-Specific Language for High-Performance Computing.**" University of Utah, UUCS-12-003, 2012.

[Oberhuber2010] T. Oberhuber, A. Suzuki, and J. Vacata. "**New row-grouped csr format for storing the sparse matrices on GPU with implementation in CUDA.**" *arXiv preprint arXiv:1012.2270* (2010).

[Pearce2010] R. Pearce, M. Gokhale, and N. Amato. **"Multithreaded asynchronous graph traversal for in-memory and semi-external memory."** In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-11. IEEE Computer Society, 2010.

[Pichel2012] J. Pichel, F. Rivera, M. Fernández, and A. Rodríguez. "**Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs.**" *Microprocessors and Microsystems* 36, no. 2 (2012): 65-77.

[PowerGraph2012] Gonzalez, Joseph E., Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "**PowerGraph: Distributed graph-parallel computation on natural graphs.**" In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 17-30. 2012.

[Pregel2010] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. "**Pregel: a system for large-scale graph processing.**" In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135-146. ACM, 2010.

[SC13] Q. Meng, A. Humphrey, J. Schmidt, M. Berzins. **"Investigating Applications Portability with the Uintah DAG-based Runtime System on PetaScale Supercomputers,"** SCI Technical Report, No. UUSCI-2013-003, *SCI Institute, University of Utah*, 2013. To appear in Supercomputing 2013 Proceedings November 2013.

[Schreiber2000] T. Schreiber, Thomas. "**Measuring information transfer.**" *Physical review letters* 85, no. 2: 461, 2000

[Soman2010] J. Soman, K. Kishore, and P. Narayanan. "**A fast GPU algorithm for graph connectivity.**" In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1-8. IEEE, 2010.

[SPARQL2008], E Prud'hommeaux and A Seaborne. "**SPARQL Query Language for RDF**", W3C, 2008.

[Stanton2012] S. Isabelle and G Kliot. **"Streaming graph partitioning for large distributed graphs."** In Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1222-1230. ACM, 2012.

[Stutz2010] P. Stutz, A. Bernstein, and W. Cohen. "**Signal/collect: Graph algorithms for the (semantic) web.**" In *The Semantic Web–ISWC 2010*, pp. 764-780. Springer Berlin Heidelberg, 2010.

[Sung2012] I.-J. Sung, G. D. Liu, and W.-M. W. Hwu, "**DL: A data layout transformation system for heterogeneous computing.**" Innovative Parallel Computing (InPar), 2012.

[Totem2012] A. Gharaibeh, L. Costa, E. Santos-Neto, and M. Ripeanu. "**A yoke of oxen and a thousand chickens for heavy lifting graph processing.**" In Proceedings of the 21st international conference on Parallel architectures and compilation techniques, pp. 345-354. ACM, 2012.

[Tsourakakis2009] C. Tsourakakis, M. Kolountzakis, and G. Miller. "**Approximate triangle counting.**" *arXiv preprint arXiv:0904.3761*, 2009.

[Vastenhouw2005] B. Vastenhouw, and R. Bisseling. **"A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication."** SIAM Review, Vol. 47, No. 1 : pp. 67-95, 2005.

[Vázquez2012] F. Vázquez, J. Fernández, and E. Garzón. "**Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach.**" *Parallel Computing* 38, no. 8 (2012): 408-420.

[VerSteeg2012] G. Ver Steeg, A. Galstyan. "**Information transfer in social media.**" In *Proceedings of the 21st international conference on World Wide Web*, pp. 509-518. ACM, 2012.

[WolfHPC] Q. Meng, A. Humphrey, M. Berzins. **"The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System,"** In *Digital Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis*, Note: *SC'12 –2nd International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC 2012*, pp. 2012.

[Wood2012] Wood, Peter T. "**Query languages for graph databases.**" *ACM SIGMOD Record* 41, no. 1, 50-60, 2012.

[XSEDE2013] Q. Meng, A. Humphrey, J. Schmidt, M. Berzins. **"Preliminary Experiences with the Uintah Framework on Intel Xeon Phi and Stampede,"** In *Proceedings of the 2nd Conference of the Extreme Science and Engineering Discovery Environment (XSEDE 2013)*, pp. (to appear). 2013.

[Xu2010] S. Xu, H. Lin, and W. Xue. "**Sparse matrix-vector multiplication optimizations based on matrix bandwidth reduction using NVIDIA CUDA.**" In *Distributed Computing and Applications to Business Engineering and Science (DCABES), 2010 Ninth International Symposium on*, pp. 609-614. IEEE, 2010.

[Zhao2013] X. Zhao, A. Chang, A. Sarma, H. Zheng, and B. Zhao. "**On the Embeddability of Random Walk Distances.**" *Proceedings of the VLDB Endowment* 6, no. 14 (2013).

[Zhong2013]. J. Zhong, and B. He. **"Medusa: Simplified Graph Processing on GPUs."** IEEE Transactions on Parallel and Distributed Systems, Volume PP, Issue 99, 2013.

LIST OF ACRONYMS

BFS	Breadth First Search
CAS	Compare And Swap
CC	Consensus Clustering
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random Access Memory
DSL	Domain Specific Languages
GAS	Gather and Scatter
GFLOPS	Giga FLoating-point Operations Per Second
GPU	Graphics Processing Unit
GTEPS	Giga Traversed Edges Per Second
ORNL	Oak Ridge National Laboratory
POC	Proof Of Concept
PR	Page Rank
RAM	Random Access Memory
RDF	Resource Description Framework
SPARQL	Simple Protocol and Resource Description Framework Query Language
SSSP	Single Source Shortest Path